Visualization of Mutation Test Data to Aid in Test Prioritization

UNDERGRADUATE HONOURS THESIS
FACULTY OF SCIENCE (COMPUTING SCIENCE)
UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Benjamin Waters

Supervisors:
Dr. Jeremy S. Bradbury
Dr. Christopher Collins

April 8, 2012

Abstract

As agile development becomes more refined and implemented, a great deal of research is conducted on how best to optimize the process. One important feature of agile development is the continuous integration and developer rhythm made possible by ten-minute build environments. These ten-minute builds however require a significant effort to be placed on continued improvement and optimization of the test suite to ensure new code is properly tested. Test prioritization thus becomes an important problem in need of consideration. We are both attempting to maximize the coverage of our tests, as well as run more important tests earlier in the process to attempt to ensure failures occur more rapidly. Approaches to solve this prioritization vary over the spectrum of fully manual selection, fully automated algorithms, and mixed approaches. This thesis aims to aid this task by offering a visualization to aid in the semi-automatic prioritization of the test suite. Through this we can decrease the cost of implementing agile ten-minute builds, and thus enable continuous feedback for the developer.

Table of Contents

List of Figures

1. Introduction

The objective of this thesis is to explore how visualization techniques can be used to aid in test prioritization, and test suite optimization. We hope the software designed can be extended upon to form a complete environment for test suite optimization. With a large set of test data it can be difficult to optimize the set of all tests for a 10 minute build. It is not simply a matter of selecting which tests to run, but also the order in which they are run. Not only do we want to detect as many potential points of error as possible, but we also want to detect as many different points of error as soon as possible. In this way we can attempt to test the code thoroughly and detect as many errors as possible at the start of the test suite. Given the impossibility of exhaustive testing in a 10 minute build, the focus is instead placed on raising our confidence to its maximum potential that the greater majority of bugs will be detected.

1.1 Problem Statement and Motivation

Test prioritization is an important issue in software testing, in particular for an agile environment involving Ten-minute Builds. This issue presents several problems, we must ensure that our selected tests detect as many issues as possible but we must also put the emphasis on early detection so that any problems occur closer to the start of the test cycle as opposed to the end. While work has been done in the past to fully automate the prioritization of tests, there has been little work in the area of visualization to allow for a user to make decisions on the prioritization. This is due to early limitations with mutation testing involving a large portion of manual work. The focus had been extensively placed upon the automation of these tests, and on an imperative to remove the onus of mutation testing knowledge from developers who simply want effective tests [1]. While in general this is a valid point and worthy of focus, this approach neglects many human computer interaction issues which must be considered for a practical optimization solution [2]. Visualizing this Test Prioritization will allow the user to express any input or insights upon the selection, which a fully-automated selection can not do.

1.2 Goals

The goal of this thesis is to design a prototype visualization capable of prioritizing tests in a test suite based on mutation test data. To allow for code manipulation as well as test usage, the system must allow for the editing of test and code. While a fully automatic test prioritization could be useful, the potential exists for the user to have insight into the problem which the system does not, if for example a certain test has repeatedly detected errors, while other 'higher value' tests have not. Thus the user must be able to reorder tests in order to override automatic test prioritization. We will be targeting a 10-minute build goal, and thus optimization of which tests will be selected is also a key issue which must be addressed. To this end we will examine how best to balance the automation of machine efficiency, and the expertise of the human developer.

Therefore our Goals are:

- Automate the system to suggest to the developer a test selection order
- Allow for the manipulation of Test Data for Prioritization
- Integrate in environment allowing code viewing and change
- Test Suite Optimization for 10 minute builds

1.3 Stakeholders

This thesis has been influenced based upon the people who will use it. If the visualization does not serve to clarify the large sets of data involved, then we have failed. Human comprehension and intuition of the data is what will be the driving force behind this thesis. Developers with a knowledge of the code and the test suite will be our primary stakeholders. Additionally a rudimentary knowledge of mutation testing would be beneficial to the developer to insure they comprehend the underlying data. Their aim is to quickly evaluate this large set of data, and manipulate it, and to this end we must make the process as painless and enjoyable as possible. This in turn leads to the use of this prioritized test suite by the developers integrating changes to the code base, and so these developers would become secondary stakeholders. With our Stakeholders identified we are able to proceed and evaluate the Thesis based on their unique requirements.

1.4 Thesis Overview

In section 2, we will be looking at some background subject material, as well as related works in the field. These related works will be in mutation testing, software visualization as well as some introductory information on ten-minute builds and human-in-the-loop optimization. Section 3 will cover the design decisions, ranging from the original work that was started with, through an early prototype and ending with the current system. The early prototype differs significantly from the ending system, and so the reasoning behind the design change will be explained. Section 4 will cover the Implementation of the system, where I will discuss the platform selected, the data structures used, and the overall structure of the code. Section 5 will conclude the Thesis with a summary detailing what goals were achieved with this thesis and suggestions for directions future work could take to expand upon this system. Finally a list of all references is included for a view of some of the tools and sources which were used.

2. Background and Related Work

This thesis is the merging of several different concepts, and so I will attempt to give an explanation of the underlying theory of each of these pieces so that the overall purpose is more evident. These topics range from mutation testing, software visualization, ten-minute builds, human-in-the-loop optimization and programs which share a related interest to this thesis. These related programs form an important guide when dealing with this problem even if the goals and methods differ.

2.1 Mutation testing

Mutation testing is the process of modifying a program's source or bytecode, in order to evaluate a test suite. The purpose of this is to either locate weaknesses in the test data or to aid a tester in developing effective tests. Modifications of the source code occur by use of mutation operators, which define the differing ways in which the original code can be changed. Each mutant version of the code seeds only a single mutation into the original, however potentially thousands of mutations can be created, resulting in a large set of data related to possible errors which may not be detected. Extensive research has been done in the subject, including many different tools being developed. The tools however vary in the goals they attempt to accomplish through mutation testing, as well as the mutation operators which they use. For the purpose of my thesis, a tests value will be evaluated based on the rate of detection of mutations and the goal will be placed upon mutation coverage as opposed to code coverage. This allows the developer to mutate only sections of code which are of interest and thus the optimization will be focused upon the selected area.

2.1.1 Example Mutation

Many different mutation operators exist but all follow the same basic principle. The source code is copied and a single error is seeded into the code. An example of the **j**ava **s**tatic **i**nsertion operator, or jsi for short, is provided in the code sample below:

Original Code

```
public class Foo{
        private String bar;
        …
}
```

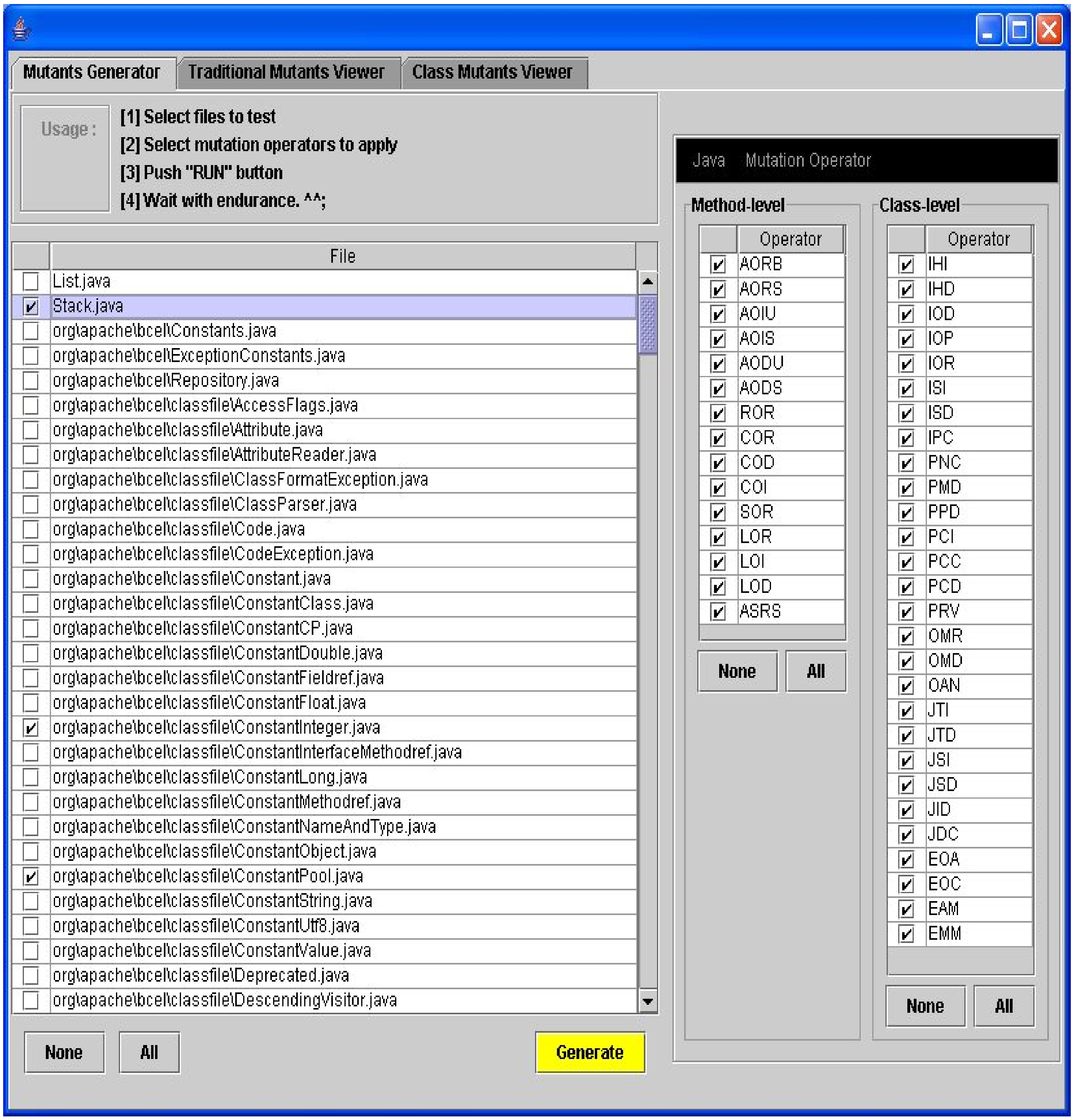
Mutated Code

```
public class Foo{
        private static String bar;
        …
}
```

As you can see in this mutation, the static keyword is inserted on the String bar. For a test to detect this mutant, output would have to differ from the expected output already received from the original code.

2.1.2 μJava

μJava[3], originally known as JMutation, is a mutation system for java programs, which includes both a mutation generator, and a test evaluation system. μJava is of particular importance to this project, as its integration was a part of the final implementation. μJava boasts a relatively large set of mutation operators with both class and method level mutations. All mutation operators can be grouped into one of six unique categories, Access Control, Inheritance, Polymorphism, Overloading, Java-Specific Features, and Common Programming Mistakes. One such example is the Modifier method change mutation, which attempts to replace modifier methods within a class with other compatible modifiers. Compatibility is determined based on a similar method signature with only a varied method name. μJava allows for the selection of each mutation operator, as well as each java source file, so if only a specific area is of interest, the mutation can be more selective. To detect this mutant thus, the test will need to produce incorrect output as a result of calling the wrong method and modifying the class incorrectly.



Usage :
[1] Select files to test
[2] Select mutation operators to apply
[3] Push "RUN" button
[4] Wait with endurance. ^^;

	File
☐	List.java
✓	Stack.java
☐	org\apache\bcel\Constants.java
☐	org\apache\bcel\ExceptionConstants.java
☐	org\apache\bcel\Repository.java
☐	org\apache\bcel\classfile\AccessFlags.java
☐	org\apache\bcel\classfile\Attribute.java
☐	org\apache\bcel\classfile\AttributeReader.java
☐	org\apache\bcel\classfile\ClassFormatException.java
☐	org\apache\bcel\classfile\ClassParser.java
☐	org\apache\bcel\classfile\Code.java
☐	org\apache\bcel\classfile\CodeException.java
☐	org\apache\bcel\classfile\Constant.java
☐	org\apache\bcel\classfile\ConstantClass.java
☐	org\apache\bcel\classfile\ConstantCP.java
☐	org\apache\bcel\classfile\ConstantDouble.java
☐	org\apache\bcel\classfile\ConstantFieldref.java
☐	org\apache\bcel\classfile\ConstantFloat.java
✓	org\apache\bcel\classfile\ConstantInteger.java
☐	org\apache\bcel\classfile\ConstantInterfaceMethodref.java
☐	org\apache\bcel\classfile\ConstantLong.java
☐	org\apache\bcel\classfile\ConstantMethodref.java
☐	org\apache\bcel\classfile\ConstantNameAndType.java
☐	org\apache\bcel\classfile\ConstantObject.java
✓	org\apache\bcel\classfile\ConstantPool.java
☐	org\apache\bcel\classfile\ConstantString.java
☐	org\apache\bcel\classfile\ConstantUtf8.java
☐	org\apache\bcel\classfile\ConstantValue.java
☐	org\apache\bcel\classfile\Deprecated.java
☐	org\apache\bcel\classfile\DescendingVisitor.java

Java Mutation Operator

Method-level

	Operator
✓	AORB
✓	AORS
✓	AOIU
✓	AOIS
✓	AODU
✓	AODS
✓	ROR
✓	COR
✓	COD
✓	COI
✓	SOR
✓	LOR
✓	LOI
✓	LOD
✓	ASRS

Class-level

	Operator
✓	IHI
✓	IHD
✓	IOD
✓	IOP
✓	IOR
✓	ISI
✓	ISD
✓	IPC
✓	PNC
✓	PMD
✓	PPD
✓	PCI
✓	PCC
✓	PCD
✓	PRV
✓	OMR
✓	OMD
✓	OAN
✓	JTI
✓	JTD
✓	JSI
✓	JSD
✓	JID
✓	JDC
✓	EOA
✓	EOC
✓	EAM
✓	EMM

Figure 1: ***µJava*** *mutant Generation System* [3]

2.1.3 Javalanche

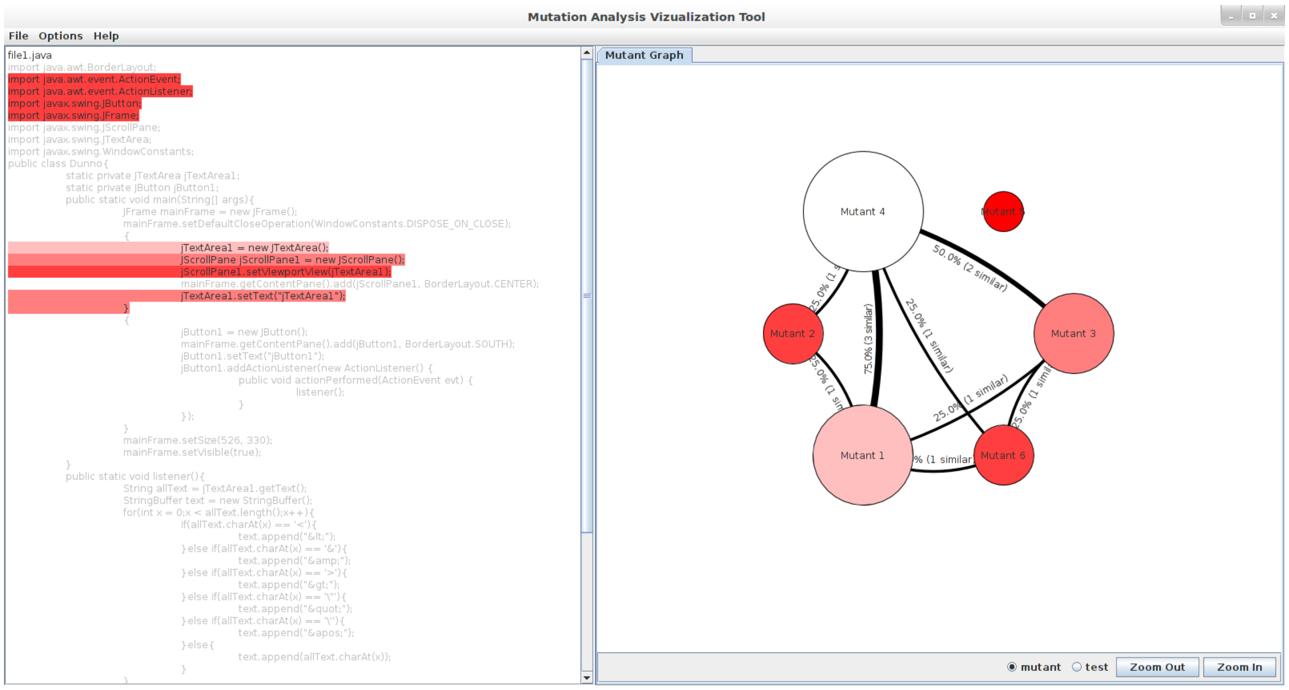
Javalanche[5] is a mutation testing tool for java source code with a focus on automation and a boasts small percentage of equivalent mutation resulting in fewer false positives. Javalanche does not use a graphical interface and is instead run from the command line. Javalanche creates custom xml files to store results which make it more convenient when attempting to automate comparisons. Javalanche was integrated with the early prototype work, however it was not used in the final system due to the fact fewer mutants were being generated then what was considered acceptable.

2.2 Software Visualization

Software visualization is the use of graphics to visually aid in the interpretation or exploration of a set of data related to some form of software metric. The success of a software visualization is dependent upon its ability to refine the raw data into something intuitive to a human user. While there has been limited work with regards to mutation visualization, it is important to describe a previous project which created a visualization, though it did not share the goal of this thesis.

2.2.1 VisMAn

VisMAn[4] was a project created by a former computer science student at UOIT. The purpose of the visualization was to create a general visualization, which would allow for the exploration and understanding of mutation test data. It consisted of two visual elements, the code element displayed on the left, and the graph element displayed on the right. The code element highlighted the locations of mutations with a colour shade of red indicating the detection rate of the mutant. The graph view was able to focus on both mutation data and test data. In either case the focused objects are referenced by nodes appearing on the graph. A larger whiter node represents that a mutant was detected more, and by contrast a smaller red node denotes that a mutation was detected less. It is important to note that the nodes colours are reflected in the code display with matching shades of red. Edges between nodes are used to denote a similarity between the two nodes, as well as a percentage indicating the numerical value of this similarity. Filtering is possible by selecting a node, which will then create a new tab. In this tab there will be a filtered version of the graph which displays the filtered node as well as any other nodes which share a connection with it. While the VisMAn project ultimately does not share the same goal as this thesis, it is important to note as it is one of very few attempts to use visualization to better understand mutation testing. See figure 2 below for a view of the VisMAn system functioning.



```
file1.java
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.WindowConstants;
public class Dunno{
        static private JTextArea jTextArea1;
        static private JButton jButton1;
        public static void main(String[] args){
                JFrame mainFrame = new JFrame();
                mainFrame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
                {
                        jTextArea1 = new JTextArea();
                        JScrollPane jScrollPane1 = new JScrollPane();
                        jScrollPane1.setViewportView(jTextArea1);
                        mainFrame.getContentPane().add(jScrollPane1, BorderLayout.CENTER);
                        jTextArea1.setText("jTextArea1");
                }
                {
                        jButton1 = new JButton();
                        mainFrame.getContentPane().add(jButton1, BorderLayout.SOUTH);
                        jButton1.setText("jButton1");
                        jButton1.addActionListener(new ActionListener() {
                                public void actionPerformed(ActionEvent evt) {
                                        listener();
                                }
                        });
                }
                mainFrame.setSize(526, 330);
                mainFrame.setVisible(true);
        }
        public static void listener(){
                String allText = jTextArea1.getText();
                StringBuffer text = new StringBuffer();
                for(int x = 0;x < allText.length();x++){
                        if(allText.charAt(x) == '<'){
                                text.append("&lt;");
                        }else if(allText.charAt(x) == '&'){
                                text.append("&amp;");
                        }else if(allText.charAt(x) == '>'){
                                text.append("&gt;");
                        }else if(allText.charAt(x) == '\"'){
                                text.append("&quot;");
                        }else if(allText.charAt(x) == '\''){
                                text.append("&apos;");
                        }else{
                                text.append(allText.charAt(x));
                        }
                }
```

Figure 2: *VisMAn, Mutation Visualization* [4]

2.3 Ten-Minute Builds

In agile development, the concept exists of a ten minute build. In such a build, the code would be able to be built and tested in a ten minute period. This should be run many times a day, either due to critical events such as checking source code into a repository, or at scheduled intervals throughout the day. This build needs to function quickly as it must be run often, and so a significant effort must be placed on continual improvement and optimization. This thesis aims to help alleviate this effort and ensure that the maintenance of this process is as enjoyable as possible.

2.4 Human-in-the-Loop Optimization

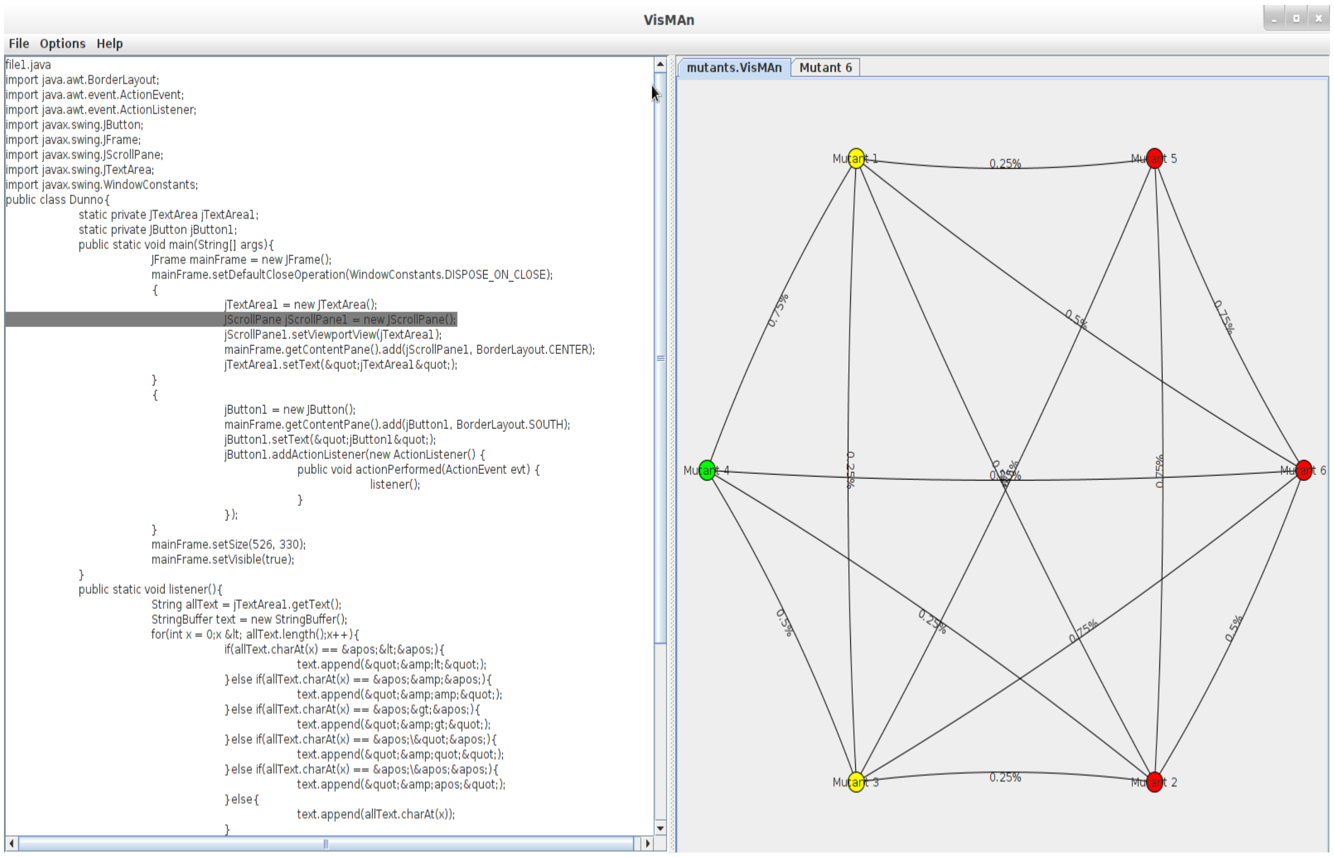
Human in the loop optimization references the problem of optimization once the human becomes an integral component of the system. The overall concept is to achieve the best of both worlds, with the superior intelligence of a human coupled with the computational speed of a machine. Maintaining the human as a component for the optimization of course brings with it the negative aspects that the end results inevitably are dependent on the human involved making them less replicable, and the process will ultimately take longer when compared to a fully automatic process in which machines make all decisions. The advantages of such a setup is the full ability to leverage human expertise on an issue, as well as not attempting to simplify overly complex sets of data to form an optimization. Several different types of human-in-the-loop optimizations exist, however the most relevant to this thesis is allowing the user to manually modify algorithmically suggested solutions with relatively no restrictions.

3. Design Approach

To determine the functionality and data structures that would be required, we gathered information from previous mutation testing tools, such as µJava and Javalanche, as well as a previous mutation testing visualization, VisMAn. Once this research was completed, we decided to utilize similar mutation test data to µJava and several key features from other tools. One of the features we determined was vital was a close integration between the visual interpretation and the raw source code.

3.1 Prototype Work

As an initial starting point several prototypes were created to explore the ideas presented by the VisMAn project, and reflect upon the challenges involved in mutation testing. A similar approach was taken, with a code display view and a graph view however differences did occur in the implementation. A greyscale highlighting replaced the red scale which was used in the original system. Likewise the colouring of the nodes differed instead based upon a colour scale to indicate the degree to which a mutation is detected without altering the size of the nodes. Selection of individual nodes functioned in a similar way, opening a new filtered graph in a tab. The focus of these prototypes however was placed on the mutants themselves, and the option to focus on tests as the primary sources of data was removed. Figure 3 depicts this prototype visualizing the same mutation data as figure 2 above.

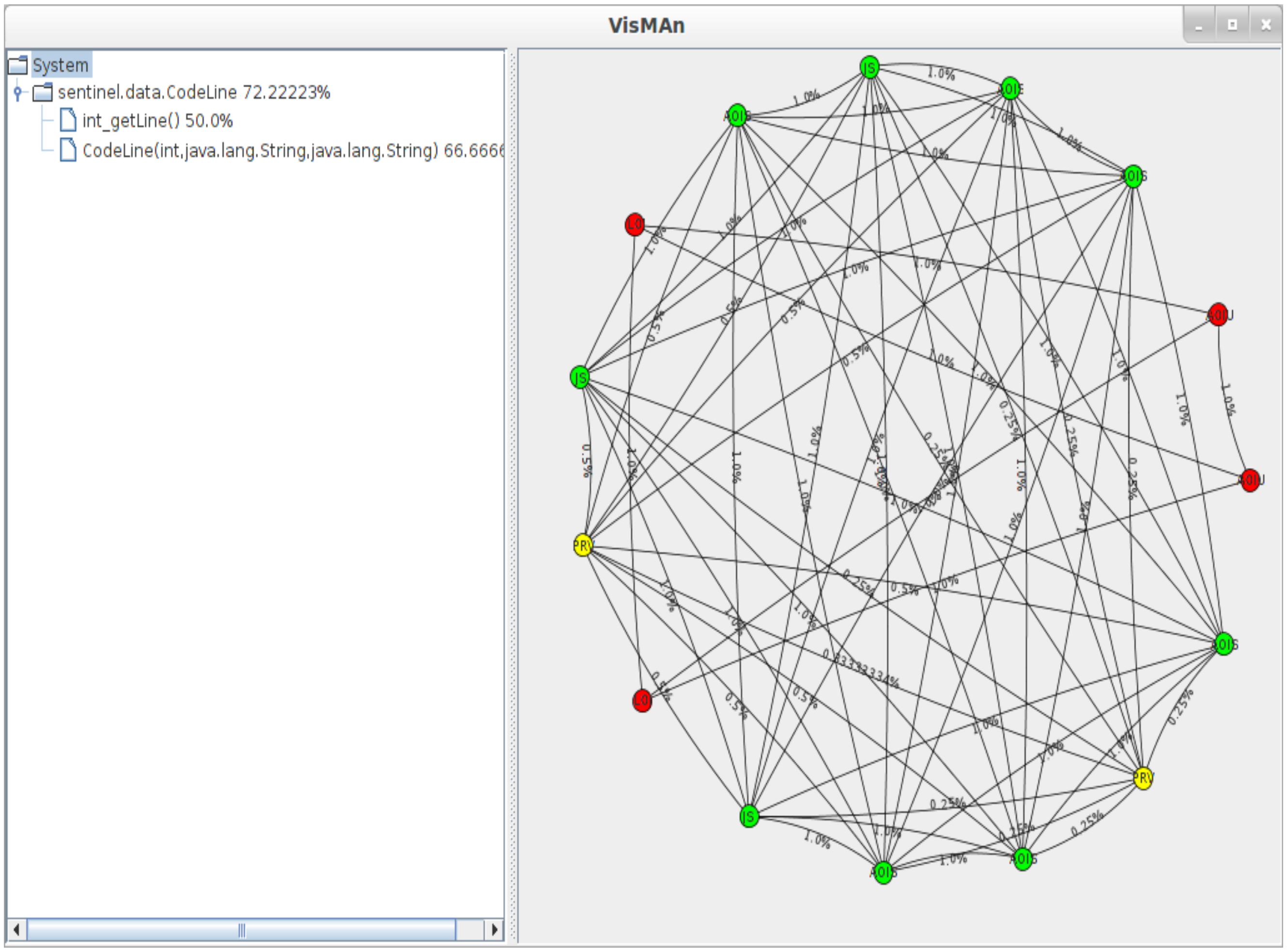


```
file1.java
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.WindowConstants;
public class Dunno{
        static private JTextArea jTextArea1;
        static private JButton jButton1;
        public static void main(String[] args){
                JFrame mainFrame = new JFrame();
                mainFrame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
                {
                        jTextArea1 = new JTextArea();
                        JScrollPane jScrollPane1 = new JScrollPane();
                        jScrollPane1.setViewportView(jTextArea1);
                        mainFrame.getContentPane().add(jScrollPane1, BorderLayout.CENTER);
                        jTextArea1.setText(&quot;jTextArea1&quot;);
                }
                {

                        jButton1 = new JButton();
                        mainFrame.getContentPane().add(jButton1, BorderLayout.SOUTH);
                        jButton1.setText(&quot;jButton1&quot;);
                        jButton1.addActionListener(new ActionListener() {
                                public void actionPerformed(ActionEvent evt) {
                                        listener();
                                }
                        });
                }
                mainFrame.setSize(526, 330);
                mainFrame.setVisible(true);
        }
        public static void listener(){
                String allText = jTextArea1.getText();
                StringBuffer text = new StringBuffer();
                for(int x = 0;x &lt; allText.length();x++){
                        if(allText.charAt(x) == &apos;&lt;&apos;){
                                text.append(&quot;&amp;lt;&quot;);
                        }else if(allText.charAt(x) == &apos;&amp;&apos;){
                                text.append(&quot;&amp;amp;&quot;);
                        }else if(allText.charAt(x) == &apos;&gt;&apos;){
                                text.append(&quot;&amp;gt;&quot;);
                        }else if(allText.charAt(x) == &apos;\&quot;&apos;){
                                text.append(&quot;&amp;quot;&quot;);
                        }else if(allText.charAt(x) == &apos;\&apos;&apos;){
                                text.append(&quot;&amp;apos;&quot;);
                        }else{
                                text.append(allText.charAt(x));
                        }
```

Figure 3: Early Prototype Work

In an attempt to optimize the system for larger sets of data, the code display incorporated a tree structure which allowed for a quick overall perspective of the code base. The root of the tree was the overall system, which could be expanded to display classes, methods and finally individual line data. Percentages next to each level indicated the average detection rating of the mutants located within the children of that node on the tree. Unfortunately the graph visualization did not scale well to an increasingly large sample of mutants, and with the theoretical potential for thousands of mutations, a redesign was determined to be necessary. This problem is illustrated below in figure 4 with a mutation sample size of fifteen. Additionally the goal of the original project had been mutation data exploration, which was not the main goal of this thesis, and so the new design demanded a radically different approach.



```
System
  sentinel.data.CodeLine 72.22223%
    int_getLine() 50.0%
    CodeLine(int,java.lang.String,java.lang.String) 66.666
```

Figure 4: Prototype with a larger set of data

3.2 Eclipse Plugin

The eclipse platform was selected due to its close integration for developers. As our primary stakeholders will be developers working on the test suite, it makes sense to integrate our visualization with their development environment. Given the previous prototype work it was determined that the visualization must be more effective at handling large sets of data, and because our goal is to prioritize tests for a ten-minute build, the visualization must recognize the time constraints involved. Through a continued process of discussion and experimentation, a new visualization was decided upon. In this new visualization each test is represented by a rectangle. The width of each rectangle represents the time it takes for that test to complete. The height of each rectangle represents the number of mutants which each test has detected. Additionally each rectangle is shaded in three different shades of blue. The darkest blue indicates the portion of the mutants detected which will only ever be found by the current test, given the set of tests currently being used in the build. The medium blue represents mutants which have been detected by this test first, but will be detected again at some point in the future. The light blue represents a mutation which was detected by this test, however was already detected previously in the set. The test sizes are then scaled horizontally so that the full length of the canvas represents the build time the user has deemed acceptable. Tests on the left side of the canvas are run first and the set progresses towards the right. This in effect forms a timeline showing the gradual decline of new detection rates and allows for the reordering of tests accordingly.

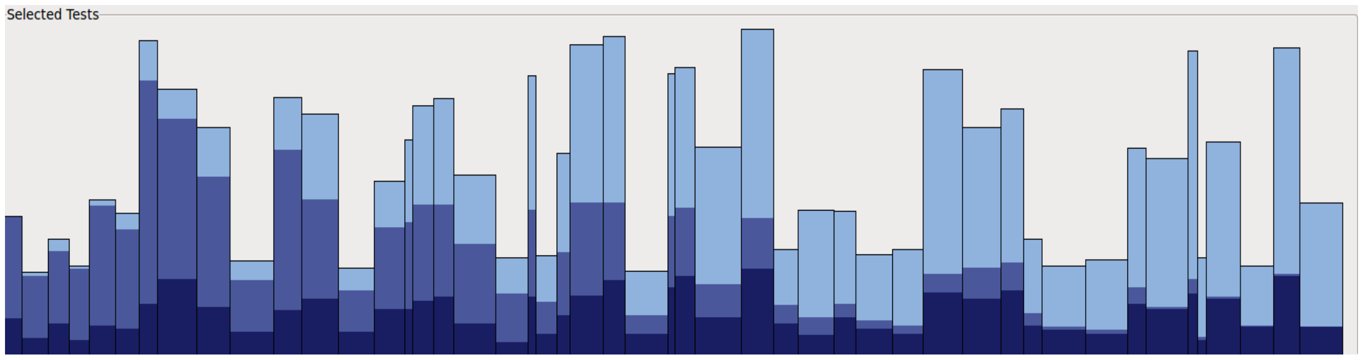


Figure 5: Tests selected to be in the ten-minute build appear in the Timeline

The user is able to select and unselect tests from the canvas by placing the mouse over the desired test and left clicking. This changes the outline of the selected test to a fusha and draws fusha highlights on tests both before and after its position based on similarity. Highlights to tests before the position indicate the sections of the already detected mutants for this test and indicate the tests which detect them first. Highlights after the current position indicate the areas in future tests which have been first detected by this test but will be detected again. This is designed to allow a user to quickly ascertain the similarity a test has with other selected tests, as uniqueness of detection becomes a key issue as the time progresses.

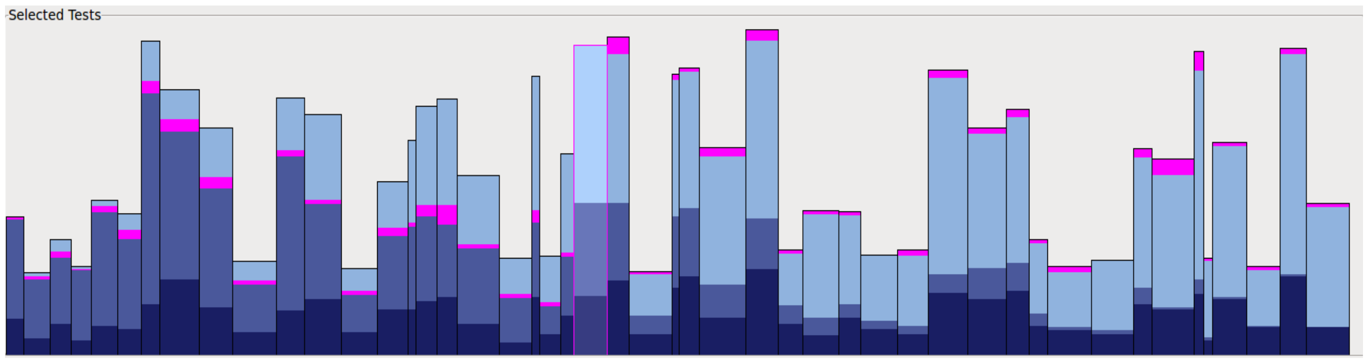


Figure 6: Selection of a test within the set highlights similarity amongst other tests

Beneath the test selection canvas is the test pool canvas. Any test which has not been selected to be a part of the timed build will be a part of the test pool. they are scaled on the same horizontal ratio as the selected tests to allow for visual comparison. The bottom test pool is not coloured with respect to any ordering in the pool. All elements in the bottom canvas are coloured to represent how they would appear if they were moved to the currently selected position and all tests after are shifted back to accommodate. In the event that no test is selected the test pool is coloured to reflect how the test would appear if it were placed at the end of the existing test set. Using test selection and right clicking the user is able to reposition tests in the pool or set to allow for any prioritization the developer wishes.

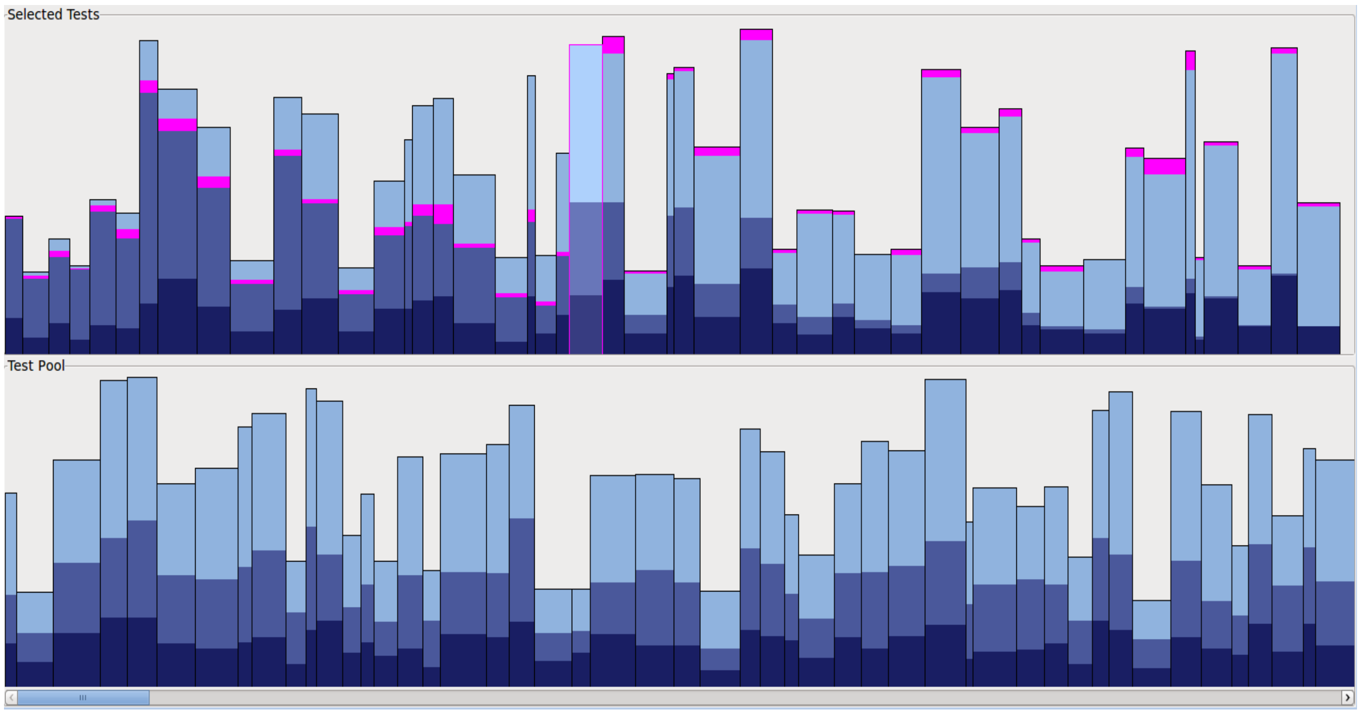


Figure 7: Tests Remaining in the Test Pool Appear Below

How best to start the selected set was also considered. It is best if these manual user changes are the exception to the rule used only when the developer has extra insight and the default starting selection is a suggestion determined algorithmically. Currently the worth of a test is evaluated through the ratio of the number of mutations which are detected over the total time of the test which is a simplification of a complex optimization problem. This is why the human interaction is such a necessary portion of the optimization, as has been previously shown in the analysis of Human-in-the-loop optimization systems [2].

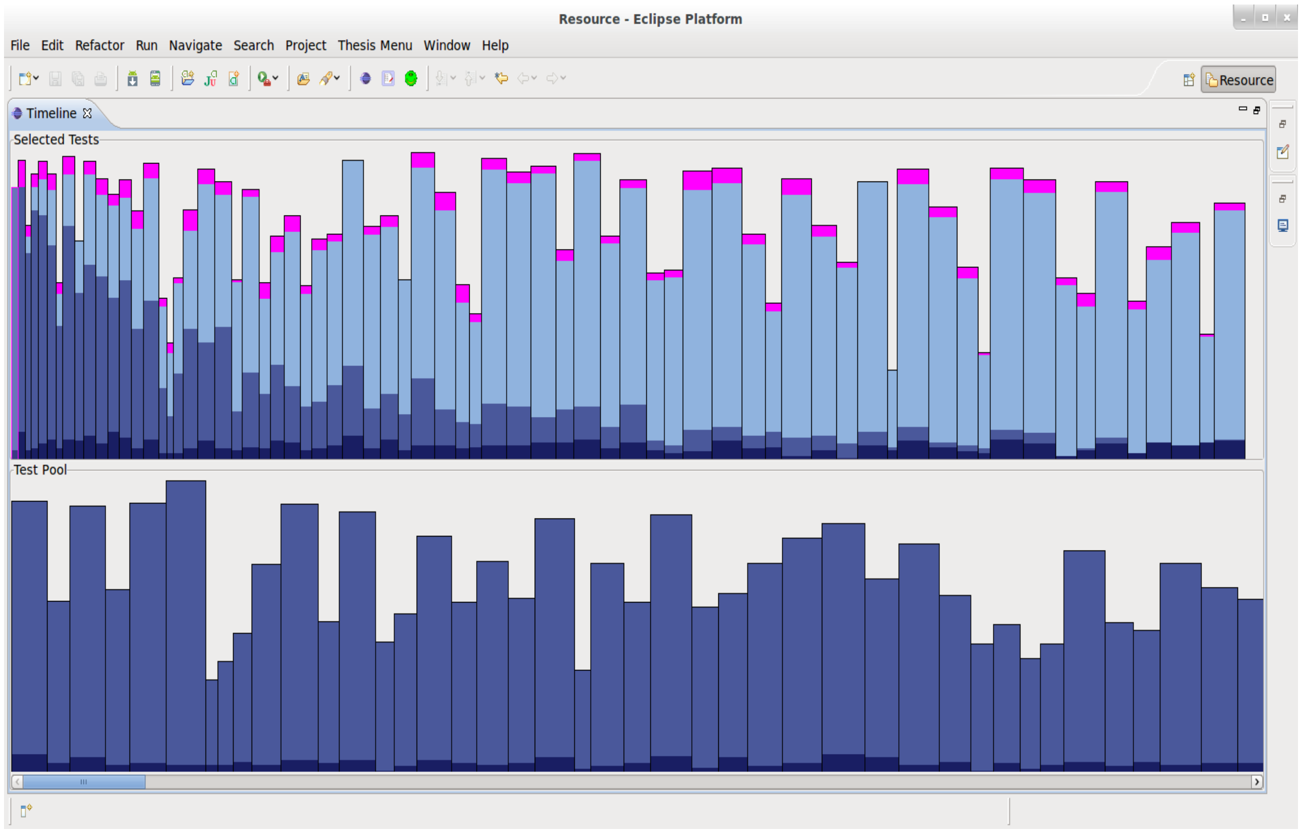
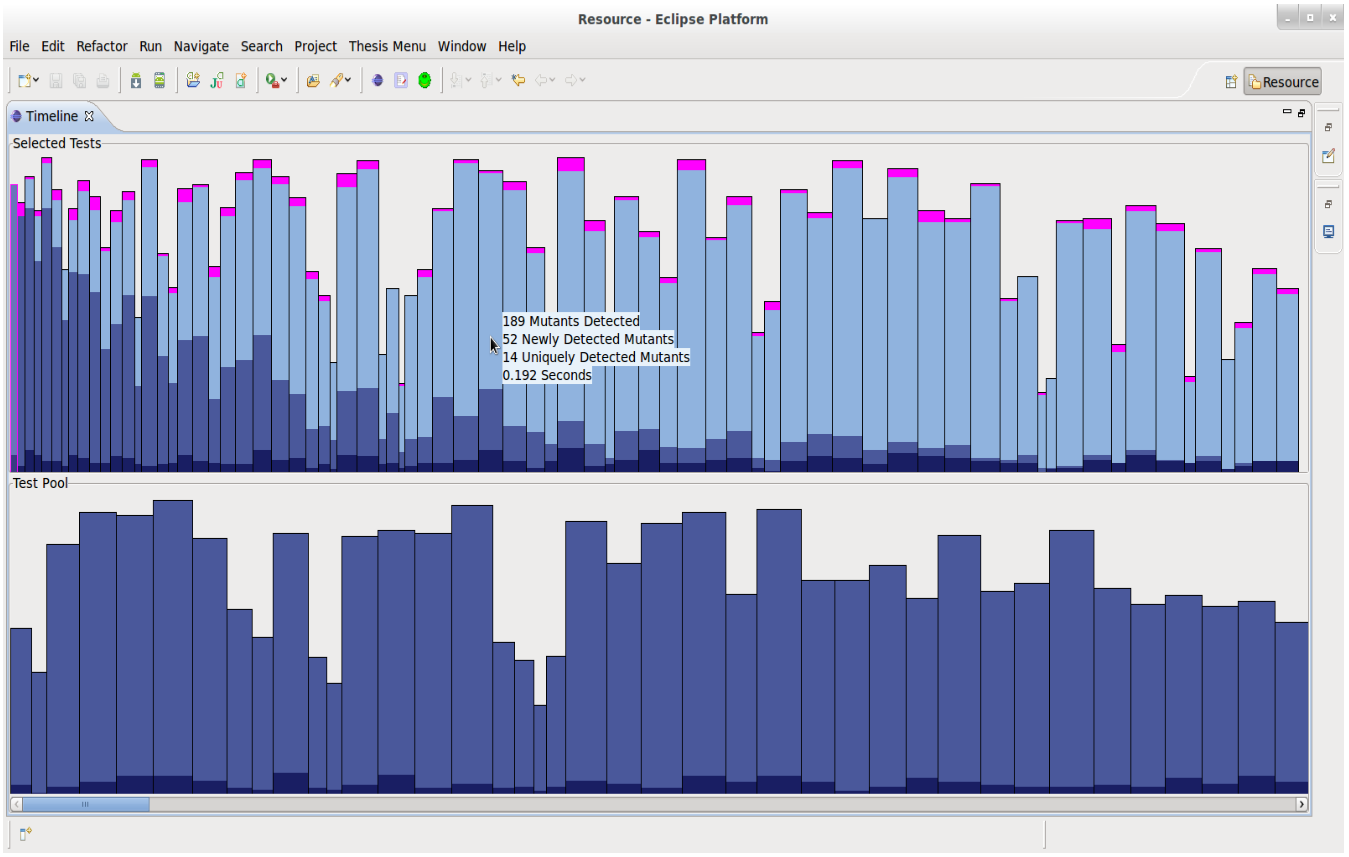

Resource - Eclipse Platform
File Edit Refactor Run Navigate Search Project Thesis Menu Window Help
Resource
Timeline
Selected Tests
Test Pool

Figure 8: Tests Sorted as an Initial Starting Point

If the mouse is hovered over a test for a sufficient amount of time, a tooltip appears detailing the raw information on the test, including detection rates and time taken. This was designed to allow the developers to examine the raw numbers involved if they so wished. This was deemed important as the visualization was intended to clarify the data but not remove it from its raw form entirely.


Resource - Eclipse Platform
File Edit Refactor Run Navigate Search Project Thesis Menu Window Help
Resource
Timeline
Selected Tests
Test Pool

189 Mutants Detected
52 Newly Detected Mutants
14 Uniquely Detected Mutants
0.192 Seconds

Figure 9: Upon a mouse hover, a tooltip appears displaying raw numbers on the test.

3.3 Example Scenario

To further emphasize the value of this approach we should consider the potential scenario of a developer using the tool. Lets say for example that through previous efforts the developer has discovered that a certain portion of code fails far more frequently than the rest. The developer starts by activating μJava and selecting only the source files which are known to be sources of failing in the previous test iterations. Once these mutations have been created and the tests are run, A suggested optimization will appear, however in this instance the developer knows through personal insight that one test which was not selected has a tendency to discover critical errors and it is currently located at the end of the test set. The developer can then reposition this test closer to the start of the session to ensure that it will be run sooner, and thus any errors will be detected much quicker.

4. Implementation

As the primary stakeholders for the application will be developers, we believe it is important to integrate our system with an existing development framework. Due to its extensibility and open nature Eclipse was selected for the final tool. The plugin is divided into three sections, the Activator, the actions, and the view.

- The Activator is the base of the eclipse plugin. It handles the lifecycle of the plugin as well as integration of all components to the plugin.
- The actions represent events which can be triggered by buttons or menu options by the user. Our plugin contains three actions the user can trigger, mutant generation, running the test suite, and modifying the time goal for the system.
- The view is used to create additional panels for content. A new view was created for the test prioritization visualization.

4.1 Data Structures

Two primary data structures exist, which are used to store all generated information for the visualization.

The first is appropriately labeled Mutant, and it stores all information related to a mutant. This includes strings for the file name and change, integer values for the line that was effected and a unique id, and finally a boolean value to maintain the state of the mutant with regards to detection.

The second is labeled TestResult and holds all the data related to how a test responded to each mutant. The stored data includes a unique integer id, double values for the time the test took to execute as well as the worth of the test, and three ArrayList of type mutant containing the detection of mutants and the degree to which they were detected. The visualization contains two lists of these test results, one list is used for all tests still in the pool and the other is used to store all tests which have been selected for the build, and thus removed from the pool list.

4.2 Mutant Generation

Mutant generation utilizes the µJava mutation generator to create the mutants from the java source code. Either the user can select a directory with data from a previous session, or the generator will be run to create new data. µJava is run by calling a bash script which creates the required µJava directories and stores the output. µJava requires the input to be located in src and classes directories, and so the program allows for the selection of the root directory for these files, and copies them as needed. Standard µJava output is a results directory, containing a directory for each class mutated. Within each of these directories are three additional directories, class_mutants, original, and traditional_mutants. The original directory contains original copies of the source and class files. The class_mutants directory contains all mutations that occur at the class wide level. Each mutant is has its modified class and source files stored in a directory, as well as an entry of what the change was in a mutation log file. The traditional_mutants directory is similar, however it focuses on method level mutations. A mutation log is stored in this directory again, however the mutations are sorted in separate directories based upon the method which it effects. Once the mutants are generated and stored in this standard format, it becomes easier to manage all the results.

A sample µJava mutation_log file contains data in the format Mutation ID : Line number : Change as shown below:

```
IPC_1:26:super( name ) is deleted
JSI_1:10:static is inserted
JSI_2:17:static is inserted
```

4.3 Running the Test Suite

The test suite root directory is specified by the user. Test suites must use the JUnit unit testing framework given the current implementation of the system. First the test suite is run against the original source code to get what should be the expected output. Once we have this baseline, the test suite is run against each mutant, and each tests result is compared against the expected result to determine when a detection occurs with the diff command. This data is all stored in a directory called tests, and is automated with bash scripts.

4.4 Visualization View

The visualization is composed of a class which extends ViewPart as well as four different listeners to handle user interaction. The ViewPart handles the organization and creation of all swt widgets. Two separate canvases need to be maintained, one to draw tests which have been selected for our ten-minute build test set, the other being all other remaining tests located in the test pool. This class holds all methods needed to shift data from one canvas to the other, as well as methods to force the view to be redrawn in the event of a change. The four listeners consist of a MouseListener, a MouseMoveListener, and two paint listeners. The MouseListener handles mouse clicks, and decides, based upon the type of mouse click, the action the user wishes to produce. Depending on the action this can involve calling methods of the view to inform it of a test change in position, or a change in selection variable to indicate a test has been focused by the user. The MouseMoveListener is used to detect the mouse's position and determine if a test is hovered over. If a test is hovered over, a tooltip appears with specific numbers for the number of mutants this test detected, as well as the time it took to run. The two paint listeners exist to draw all the visualization data to the two different canvases. This can be triggered from the ViewPart if an event occurs like a change in selection, or it can be triggered by a paint event if for example the view is resized. The listeners in general do not hold any data themselves, with the exception of the paint listeners which hold the colour information as constants.

5. Conclusion

During this research I have learned a great deal related to mutation testing, both the potential gains, and the issues involved. I have also looked extensively into visualization techniques, both directly and indirectly related to software visualization. Many tools exist to implement and report on mutation testing, however there has been little research in the field of visualization related to mutation testing. While developers continue to push the onus of mutation testing on automation, the advantage of manual control is substantial and need not be ignored. To this end we developed a system which attempts to both automate the process of test evaluation, while allowing the user to visualize the problem in a comprehensive manner as opposed to raw data output. In this thesis I researched how mutation test data could be used as the base data for a visualization which aids in the prioritization of tests within a test suite, particularly when a time constraint is imposed on the selection.

5.1 Summary

It is important to consider the original goals of the project as we evaluate its current state and future. To reiterate our focus, the goals of the project were to allow for the manipulation of test data for prioritization, integrate the visualization in an environment allowing for code and test manipulation, to automate the suggestion for a test set, and finally allowing for the optimization of a set of tests when time is a critical resource such as in a ten-minute build environment. The visualization effectively allows for the reordering of the tests, and removal of any tests which are deemed unwanted, so this goal has functionally been achieved. As the system is now an eclipse plugin, it very closely integrates with the development environment, allowing for the developer to modify tests or code and rerun the results as opposed to being forced into a more disconnected method. The sorting of the tests at the initial start serves as a recommendation to the user for how to proceed. Finally as the visualization has a strong focus on prioritization based on total time and early detection, we have heavily emphasized the importance of this resource, and it is compatible with the practice of ten-minute builds.

5.2 Future Work

We envision the current eclipse plugin and visualization to be the foundation for further extensions. In particular a custom editor which offers highlighting of mutation data on the java source files would be beneficial for the user while developing the test cases. This editor could detail points of mutation which have already been covered, as well as points of mutation which have not to aid the developer ensure mutation coverage, and thus remove points of possible failure. A visualization similar to the current package explorer would also be beneficial to allow a developer to quickly gain an overall perspective of which source files require the most attention in developing further testing. We believe these tools integrated together with the eclipse platform would create a powerful tool for test development and prioritization.

A user study would be beneficial for examining how developers use the visualization to attempt to perform a prioritization. With controlled scenarios, and test subjects who comprehend the problem involved, real use of the visualization would be incredibly valuable for future versions. Unfortunately due to the time constraints on the project a user study was not feasible, however it is a logical step in any continued development.

Additional study of automatic test prioritization would aid the tool in refining its usability. In the development of the project it was determined that the best initial selection was one optimized algorithmically with the developer then making appropriate changes given their unique insight. While currently there is a degree of optimization with regards to mutant detection against time, the problem becomes significantly more complex when dealing with variables such as new detection and unique detection with in the test set. As these values are dependent upon previous choices this optimization is nontrivial and unfortunately was not fully addressed before a feature freeze occurred. With a more accurate initial optimization we will ensure that developers making changes to the set would be the exception as opposed to the rule of its usage.

References

1. Offut, Jeff. “A Mutation Carol Past, Present and Future”, George Mason University. April 2009

2. Scott, Stacey D., Neal Lesh, and Gunnar W. Klau. *Investigating Human-Computer Optimization*. Cambridge, MA, USA: Mitsubishi Electric Research Laboratories, Dec. 2001. PDF.

3. Ma, Yu Seung, Yong Rae Kwon, and Jeff Offut. *MuJava*. Computer software. *MuJava Home Page*. Vers. 3.0. George Mason University, Nov. 2008. Web. 8 Apr. 2012. <http://cs.gmu.edu/~offutt/mujava/>.

4. Falkenham, Jeffery, and Jeremy S. Bradbury. "Visualization of Mutation Test Data."*Sqrg.ca*. University of Ontario Institute of Technology. Web. 8 Apr. 2012. <http://faculty.uoit.ca/bradbury/sqrg/projects/mutation_visualization.html>.

5. Zeller, Andreas. "Javalanche Mutation Framework." *Javalanche.org*. Saarland University, 11 June 2011. Web. 8 Apr. 2012. <http://www.javalanche.org/>.